```cpp
//-----------------------------------------------------------------------------
// nd_vector.h
//-----------------------------------------------------------------------------
#ifndef __ND_VECTOR__
#define __ND_VECTOR__

#include <initializer_list>
#include <vector>
#include <string>
#include <fstream>
#include <iostream>

namespace numeric{

    class nd_vector{
        typedef std::vector<double> container_type;
        container_type x;

        public:
            typedef container_type::value_type value_type;
            typedef container_type::size_type size_type;
            typedef container_type::pointer pointer;
            typedef container_type::const_pointer const_pointer;
            typedef container_type::reference reference;
            typedef container_type::const_reference const_reference;

            explicit nd_vector (size_type n = 0);
            nd_vector (std::initializer_list<double>);
            size_type size (void) const;
            void read (std::ifstream & input_stream);
            void print() const;
            reference operator [] (size_type);
            value_type operator [] (size_type) const;
            pointer data (void);
            const_pointer data (void) const;
    };
}

#endif


//-----------------------------------------------------------------------------
// nd_vector.cpp
//-----------------------------------------------------------------------------
#include "nd_vector.hh"

namespace numeric{

    nd_vector::nd_vector (size_type n): x (n, 0.) {}

    nd_vector::nd_vector (std::initializer_list<double> il): x (il) {}

    nd_vector::size_type nd_vector::size (void) const{
        return x.size ();
    }

    void nd_vector::read (std::ifstream & input_stream){
        std::string line;
        while ( getline (input_stream,line) ){
            x.push_back(std::stod(line));
        }
        input_stream.close();
    }

    void nd_vector::print() const{
        for (auto it = x.cbegin(); it != x.cend(); it++)
            std::cout << *it << std::endl;
        std::cout << std::endl;
    }

    nd_vector::reference nd_vector::operator [] (size_type idx){
        return x[idx];
    }

    nd_vector::value_type nd_vector::operator [] (size_type idx) const{
        return x[idx];
    }

    nd_vector::pointer nd_vector::data (void){
        return x.data ();
    }

    nd_vector::const_pointer nd_vector::data (void) const{
        return x.data ();
    }
}
```

```
//-----------------------------------------------------------------------------------------------------------------------
// dense_matrix.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef DENSE_MATRIX_HH
#define DENSE_MATRIX_HH

#include <istream>
#include <vector>

namespace la {

    class dense_matrix final{
        typedef std::vector<double> container_type;

        public:
            typedef container_type::value_type value_type;
            typedef container_type::size_type size_type;
            typedef container_type::pointer pointer;
            typedef container_type::const_pointer const_pointer;
            typedef container_type::reference reference;
            typedef container_type::const_reference const_reference;

        private:
            size_type m_rows, m_columns;
            container_type m_data;
            size_type sub2ind (size_type i, size_type j) const;

        public:
            dense_matrix (void) = default;
            dense_matrix (size_type rows, size_type columns, const_reference value = 0.0);
            explicit dense_matrix (std::istream &);
            void read (std::istream &);
            void swap (dense_matrix &);
            reference operator () (size_type i, size_type j);
            const_reference operator () (size_type i, size_type j) const;
            size_type rows (void) const;
            size_type columns (void) const;
            dense_matrix transposed (void) const;
            pointer data (void);
            const_pointer data (void) const;
    };

    dense_matrix operator * (dense_matrix const &, dense_matrix const &);
    void swap (dense_matrix &, dense_matrix &);
}

#endif // DENSE_MATRIX_HH


//-----------------------------------------------------------------------------------------------------------------------
// dense_matrix.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <sstream>
#include <string>
#include "dense_matrix.hh"

namespace la{

    dense_matrix::dense_matrix (size_type rows, size_type columns, const_reference value):
        m_rows (rows), m_columns (columns), m_data (m_rows * m_columns, value) {}

    dense_matrix::dense_matrix (std::istream & in){
        read (in);
    }

    dense_matrix::size_type dense_matrix::sub2ind (size_type i, size_type j) const{
        return i * m_columns + j;
    }

    void dense_matrix::read (std::istream & in){
        std::string line;
        std::getline (in, line);
        std::istringstream first_line (line);
        first_line >> m_rows >> m_columns;
        m_data.resize (m_rows * m_columns);
        for (size_type i = 0; i < m_rows; ++i){
            std::getline (in, line);
            std::istringstream current_line (line);
            for (size_type j = 0; j < m_columns; ++j){
                /* alternative syntax: current_line >> operator () (i, j);
                 * or: current_line >> m_data[sub2ind (i, j)];
                 */
                current_line >> (*this)(i, j);
            }
        }
    }
```

```
    void dense_matrix::swap (dense_matrix & rhs){
        using std::swap;
        swap (m_rows, rhs.m_rows);
        swap (m_columns, rhs.m_columns);
        swap (m_data, rhs.m_data);
    }

    dense_matrix::reference dense_matrix::operator () (size_type i, size_type j){
        return m_data[sub2ind (i, j)];
    }

    dense_matrix::const_reference dense_matrix::operator () (size_type i, size_type j) const{
        return m_data[sub2ind (i, j)];
    }

    dense_matrix::size_type dense_matrix::rows (void) const{
        return m_rows;
    }

    dense_matrix::size_type dense_matrix::columns (void) const{
        return m_columns;
    }

    dense_matrix dense_matrix::transposed (void) const{
        dense_matrix At (m_columns, m_rows);
        for (size_type i = 0; i < m_columns; ++i)
            for (size_type j = 0; j < m_rows; ++j)
                At(i, j) = operator () (j, i);
        return At;
    }

    dense_matrix::pointer dense_matrix::data (void){
        return m_data.data ();
    }

    dense_matrix::const_pointer dense_matrix::data (void) const{
        return m_data.data ();
    }

    dense_matrix operator * (dense_matrix const & A, dense_matrix const & B){
        using size_type = dense_matrix::size_type;
        dense_matrix C (A.rows (), B.columns ());
        for (size_type i = 0; i < A.rows (); ++i)
            for (size_type j = 0; j < B.columns (); ++j)
                for (size_type k = 0; k < A.columns (); ++k)
                    C(i, j) += A(i, k) * B(k, j);
    return C;
    }

    void swap (dense_matrix & A, dense_matrix & B){
        A.swap (B);
    }
}
```